Aging Gracefully! Writing Software that Takes Change in Stride

Introduction

The Nature of Software: Poorly Defined or Changing Requirements

Let's face it, Being responsible for creating software is a thankless job, Long hours, little recognition for your brilliance and ingenuity, and to add insult to injury, your already vague requirements arc changing faster than the O.J. Simpson defense strategy. "I've changed my mind about the interface; 1 want multi-plot graphs with dialog boxes instead, and I want you to change the output to a spreadsheet file and you also need to add the ability to send the data over the net to my I'C in real-tirne. Have a new version ready for me clay after tomorrow. " Sound familiar?

The Impact of Changing Requirements for the Unprepared

Such is the nature of software engineering, so we'd better get used to it, right? Not so fast. Ma ybe wc can't do much to teach the Powers That Be about the discipline of engineering, but what about the long hours and the brilliance part? isn't there *something* that can be done to make our programs more resistant to the abuses of fickle managers? Even a well-designed application running in a stable environment faces alteration if its virtues are live on through the wonders of code reuse. Life in LabVIEWland certainly helps in keeping up with the presto-chango needs of a dynamic project, but a poor design approach can have you scouring diagrams all night trying to propagate the effects of that last change, Or worse yet, having the angry stares of seven impatient team members burning a hole in your back as the phrase "Are you done yet?" keeps ringing in your ears.

The Value of Designing Code to Handle Change Gracefully

Now, if wc were to work smarter, not harder, we would apply our brilliant ingenuity to craft an architecture that handles weekly design changes with complete aplomb. Code reuse could become so effective for us, we'd have half the code done before the next version enters the vaporware stage. Capturing requirements into running code would proceed so smoothly, we'd get ahead of the rest of the team and have Project curtail the show to keep the software people from leading the hardware design. Sound like ridiculous fantasy? Surprise! Each of these things has actually happened in m y group at JPL's Measurement Technology Center. You can do it, too. It's all a matter of knowing how to usc your tools to implement good software engineering principles. Planning for change and few defensive design strategies can make you the project hero instead of the scapegoat.

Strategies & Techniques for Creating Flexible, Robust Code

By no means an exhaustive treatise on the subject, the following principles have been found useful in guiding code development to produce flexible, robust architectures. The overall philosopy is to take a consistent, methodical approach to diagram construct ion. Develop each panel with a thought about which elements express an essential architectural feature and which are actually just a specific parameter that might move around when the hardware manager decides he wants "the digital bus multiplexed instead and a new A/D installed with 14 bits, not 12." Modularize and document rationale as you go with the notion that three years from now you will be dragged out of your sleep in the middle of the night to come in and perform emergency surgery on YOUR code and save the company's shareholders. (Or to rescue that occasional errant spacecraft.) Think of it as "Zen and the Art of Code Maintenance."

Follow a Modular/Hierarchical Approach

One of the best architecture-level design principles to follow that will make functional changes mercifully easy to deal with and turn the code reuse holy grail into reality is to develop your application as a hierarchy of modular elements. LabVIEW is a natural for graphically capturing a set of requirements in the venerated data flow represents t ion, hierarchical bubble charts and all. Design quickly follows in either a top-down or bottom-up (or both) approach as elements resolve themselves into greater detail with the choices of data types, structures, and library VIS made at each level. Our little secret, of course, is that once we've whipped up a CASE tool product that would traditionally be passed along to the coding team for implementation, we smugly click the run button and hand the output data to the customer instead. (After all, "Real men *draw* their code," right?)

Restrict the scope of each VI

One of the most painful mistakes I see LabVIEW hackers struggle with is what I gently refer to as the "Italian style of software development" (spaghetti code). Wires going every which way and a diagram half the size of Manhattan and twice as con uplicated as the Pentagon's plumbing. Give me a break. Learn to restrict the scope of each VI to do no more than one specific task you can sum up in the one or two sentences of documentation you're dutiful l y typing into the corner of each diagram. This is the principle of *cohesion;* if it doesn't contribute to a module's function, move it to where it logically belongs.

Use the *7 +/-2* rule of thumb for limiting the complexity of panels

Psychologists have long known what the hapless developer senses in his gut when faced with modifying his wonderous LabVIEW mega-panel: Human brains are equipped to handle at most seven items at once, give or take a few. Respect this cognitive limitation and keep the complexity of your panels to a manageable level. There's no honor lost in having a VI with only three or four objects on the diagram. (Besides, these are easiest to modify and get the most reuse later.)

Avoid embedding multiple copies of the same functionality

Another common mistake 1 often see is the cookie-cutter approach to common functionality. Sure, it's easy to select a block of code in LabVIEW and drag it over to the next panel. Viola! instant copy of a debugged routine; wire it in and off we go... Such expediency has its price when mods are needed to this common code. Successful software engineering is not synonymous with making bet ter pastry. To paraphrase the old oil filter commercial, "You can pay me now or pay me later *with interest and penalties."* If you need the same process over *here,* too, then take the time to drop it into a sub-VI and maintain only one piece of code. (Word on the street has it that an up and coming vesion of LabVIEW will make this process a snap.)

Be general & generic at the bottom, Get more specific as you go up

Finally, don't modularize haphazardly; take a logical, hierarchical approach. 1 often follow both a top-down and a bottom-up approach simultaneously. At the bottom, define drivers and basic processing steps that are completely generic, containing nothing specific to your application. These will be heavily reused and should form the basis of your software reuse library. Keep that in mind as you design them. A glance at the diagrams of the lowest level VIs should answer the question, "How... " very well but the question **'Why... "** not at all. Build a layer of VIS above which are more abstract and call these as sub-VIs to provide the details. Continue on to the next layer, becoming more distant from implementation details and more specific to the needs of the application as you go up.

At the top, concentrate on a simple diagram to answer the very application-specific question "What,.." but give no clue as to "How..." Your 7+/- 2 sub-VIs here will provide more "how" details and 1osc some of the "what" flavor. Here it's important to divide your hierarchy cohesively, with the resulting trees of VI calls being as loosely coupled to each other as possible.

(It's unfortunate that the View Hierarchy panel LabVIEW 3 provides cannot be manipulated to better visualize the program structure. It would become a genuinely useful form of documentation. Perhaps NI will add this capability in LabVIEW 4?) in the middle you will meet your bottom-up products as elegant calls to carry out the processing needed above. ("Make it so, Number 1!")

The Proper Use of Globals and Enums

One of the most convenient additions to LabVIEW 3 is globals; it saves us the effort of hand-coding ersatz global function VIs in each application. Object-oriented purists sneer at the concept of a global VI in the same way structured language purists react derisively to the word "goto." Ignore these people. Globals are hereto stay and you're the big winner, especially when modification time comes around and you're in the hot seat to crank out Version 2.

The value of using globals in software design

What's the big deal with globals and how do I use them anyway? In a word, *Parameters.* in a phrase, *Front Pared controls and indicators processed in remote sub-VIs.* Globals let you parametrize on a global scale much as front panel controls and indicators let you parametrize a sub-V] call. Globals let you stuff a value into a pidgeon hole and pull it back out again at a later time, in another place. Globals area great way to deal with constants and variable parameters in your code. Because, as you will learn sooner or later, a constant is not a constant to a software task manager. And when it comes to making changes down the road, do you really remember what that "2" passed to the sub-VI really *means*?

Variables vs. symbolic & manifest constants

To understand the proper way to apply globals in your design, we need to make the distinction between a symbolic constant and a manifest constant. A symbolic constant is the meaningful symbol (read "Descriptive Native Language Phrase") which is associated with a datum that a VI interprets in performing its function. These are the values that are supposed to be constant, so you naturally hard-code them into your diagrams with one of the constant objects and then type in a value. Then you clone them all over your application] ι. You know, the same ones that you have to go back and carefully search through all your diagrams to adjust a week later because there was this *change...* Don't embed "magic numbers" in your software. I Define a global, put the value on its panel and make it the default, and clone the Read Global VI to access this value throughout your code. Give it a descriptive name so you will know exactly what it means and why it's there. The readability of your code will increase exponentially. It's like.., magic.
We need to distinguish as ymbolic constant from a variable and a manifest constant. A variable is no different from what we've described above, except that we reserve the right to change the value from time to time. In this case, we assign an initial or default value to the global during program initial ization, and write new values when and where appropriate. And lest we take this symbolic stuff a bit too far, we recognize that there are some constants which are truly constant and make the most sense when expressed as a number. Accessing the first element of an array with index O, shifting a '1' to make a bit mask, or dividing by 2 are good examples. Don't overdo a good thing.

Using Enums and Rings instead of symbolic constants

A global is not always the most appropriate choice to shi ft crypt ic numbers into meaningful symbols. If the actual values of a parameter are constant and limited in number, consider using an enum or a ring. If the values are meaningful to the code but not to the human reader, then use a digital text ring to represent the parameter. If the values are arbitrary, but only need to be unique for each case, then select an enum. The advantage of an enum really shines when you use it to control a case statement: The symbolic labels appear as each case name automagically. The downside of enums rears its ugly head if you try to extend one across an application and then have to change it later. You'll have to laborious y fi x each reference in the code. Use a

digital ring or global instead; no nifty case labels, but the numbers adjust to the change nicely. Enums are best used in a local scope (i.e., within a single subVI).

Text string globals for dialogs & panels

One slick usc of globals is to symbolically reference text strings used in dialogs, on front panels, and in files. Create a text string global with a control for each string, type in the text for each, and make them the default values. Use the string in the program by reading the global instead of a string constant, You have all your strings in one place, allowing you to make quick and complete changes to move from development to implementation versions, adapt the code to a new application, or even to change languages for an multi] ingual delivery.

Use of Arrays & Clusters

An important technique for generating flexible code is to make intelligent use of LabVIEW's array and cluster capabilities. Most of us are familiar with array basics, but not everyone has figured out clusters yet.

The value of LabVIEW's variable-size arrays

Structuring your data to naturally capture an expression in array form is almost always a good idea. LabVIEW excels in processing data in the form of an array, with many handy library functions for manipulating array data. Learn what they are and how to usc them. You will then start seeing your data and structures in these forms and naturally chose the best architectures. Diagrams and structures will be simple and concise, and be easier to modify. A valuable aspect of LabVIEW's arrays is their variable sizing. Leverage this characteristic to make your code insensitive to changes through expression in array form and the avoidance of any hard-coded array sizes. Many times I have had to alter the sizes of structures on a global scale, or even worse, had to have the code do so on the fly (dynamic allocation) and didn't even think twice about whether my 120 or so sub-VIs would tolerate such a change. 'l'hey just... worked. I played.

Using clusters for encapsulation of disparate data types

Clusters arc used to encapsulate a cohesive set of disparate data types, which are stored, processed, or transported as a group. Clusters reduce clut tcr and visual complexity, especial] y when it comes to wiring icons together. Think of thcm as way to hierarchically collapse a bundle of wires into one wire, much like you do with structuring sub-Vis. The key to using thcm successfully is to keep them highly cohesive and loosely coupled. If you're packing the kitchen sink into your clusters, you're abusing the concept. If you find yourself splitting off parts of different structures and combining them to get your data processed, you need to revisit your design.

Why clusters should (almost) always be referenced by name

Most developers that I see who are using clusters are using the degenerate form, the basic cluster, to bundle and unbundle their data. A much safer (and better documented) approach is to use the bundle and unbundle by name VIs instead. This is especially true in a changing code environment, since accessing your cluster fields by name does not need any modification if the cluster later has elements added or removed. This approach also reduces diagram clutter by allowing you to show only the cluster elements that are relevant. It improves readability when you use a meaningful phrase name for the fields, And it provides for an elegant means of implementing a parametric architecture for data more con plicated than a simple global. And as with manifest constants, there will always be the occasional case where an unnamed cluster is the most sensible choice.

Parametric Architectures

Parametric what? What is a parametric architecture and should I care? Simply put, it is the complete application of the principles presented here, with the changeable elements located in one place, such as a file or a global panel. You should care because employing this design technique can provide some of the greatest insensitivity to change and it makes code much more readible. It also allows for practical code reuse of more complex routines and opens the door for more advanced behavioral tricks such as parameter files, scripting, and table-driven architectures.

How parametric design works

A parametric design presumes that all the symbolic constants, array sizes, files referenced, text strings displayed, and even sequences of code executed may be called upon to vary in value, size, amount, order of execution, etc. These choices may depend upon decisions made at compile time by a manager, at run time by an operator, or in response to data acquired during execution. By controlling these characteristics through parameters, restructuring at this level involves little more than adjusting the value of a few parameters and re-running the code with little concern for its impact. That is, other than to absorb the *"wow"* and *"genius!"* comments you'll get from the team members who thought they'd just torpedoed your weekend plans with a new spec to recode.

Configuration & Default Parameter Files

Once a parametric approach has been chosen, some valuable programming tricks become available. Using files to store configuration parameters and default values is one of them. Storing the parameters that control the internal structure of your application in a file means that changes can be made to the program without editing the VIs, a move which tends to put software managers' minds at ease. It also naturally makes the code more resusable. Many applications in the data acquisition world have remarkable similarities which don't really need to be reinvented for every new task. They just need to be built once, intelligently, and then have their parameters tweaked to suit the next job.

Table-Driven or Script-Driven Behavior

A more advanced technique carries this concept further. More sophisticated behavioral control can be exercised without code modifications through the use of table-driven or script-driven architectures. This means using LabVIEW to develop a language which interprets a table or a text file containing parameters and procedural elements to carry out its tasks. It amounts to trading the general but unstructured nature of the LabVIEW environment for one that only knows how to carry out common primitive tasks but needs less programming to do so. We are having great success using these architectures to allow system testers to write, run, and modify their own procedures with little risk of software rewrites to handle their next brilliant test idea.

Defensive Architecture Strategy

Without sounding paranoid, a lot of what goes into making an architecture resistant to change is to take a defensive attitude with the customer. After all, the farther along a project gets when a boo-boo is discovered and needs recoding, the more costly it becomes. Spending a little extra time on your part to design in a safety net not only provides valuable insurance down the road, but also gives you a satisfying outlet for all that pent-up brilliant ingenuity you've been wanting to express.

Be familiar with the application domain

One way to get an edge on controlling the destiny of your code is to be as familiar with the application domain as you can. Knowing the finer distinctions of the system your software is part of provides you with the insight you need to anticipate where the changes are likely to occur. This insight should guide your choice of architecture and assist you in picking out the

parameters. Take the attitude that if it might change, it probably will, so have the solution in your back pocket ready to go. It's not wasted effort as much as it's a good habit to follow.

Consistency, Standards, and Code Reuse

Consistency is key; Discipline leads to consistency

Some genera] principles bear mentioning here. The overall philosophy behind any successful software engineering task is consistency and discipline. Lack of either one is what gets us in trouble. Software is a powerful engineering tool, but as with anything powerful, there's the other edge that cuts in your direction, too, The flexible nature of software may make it possible to capture whatever whim hits you, but this nature also makes it easy for you to end up with a hopeless mess. Be vigilant against letting sloppiness, laziness, or expediency creep into your craft. Discipline is the foundation for consistent approaches and consistent approaches yield predictable, maintainable results, Your reputation and career rest on your results.

Use standard approaches for typical tasks

Once your mind has been trained to recognize the essential form of the various tasks and subtasks you deal with, you should begin to see things not as how they are different from one another, but how they are the same. Capitalize on this sameness to generate a code reuse library, design with parameters, and employ standards in structure, documentation, style, and appearance.

Reuse low-level modules that perform common tasks

With experience gained from abstracting the commonality among similar types of tasks, you can shrink the amount, and therefore complexity, of your code. Package the abstracted form into a sub-VI and reference it this way, in some cases even if it is only called from one location. One change covers all references, whether in structure or location. It also makes it possible to substitute emulators or stubs, partition tasks among a team, or make major behavioral changes without making a new program.

Design modules with thought to using them in other projects

Finally, design every module with the thought of having to reuse it down the road in another project. Or that you or your buddy is going to have to modify it and maintain it later. You never really know what you're going to encounter in the future, (C)K, if you do, you have an interminably boring job which you should quit immediately.)

Documentation

The last word on software engineering? Documentation. Yikes! Yes. Documentation can be your friend. Your good and only friend when you're under the gun to fix the code you wrote eleven months and 14 projects ago. "But," I hear you say, "LabVIEW is self-documenting." Yeah, right. You tell me that when you're staring at somebody else's spagetti plate special with the job of making it go. Employing the principles I've outlined above certainly makes LabVIEW *more* self-documenting, but nothing takes the place of text boxes liberally sprinkled about a diagram, explaining why certainly structures anti data types were chosen and what the consequences of modification would be. Remember: The reputation you save could be your own.